1. **Write a program to perform the following**

**An empty list**

**A list with one element**

**A list with all identical elements**

**A list with negative numbers**

**Test Cases:**

1. **Input: [] , Expected Output: []**
2. **Input: [1] ,Expected Output: [1]**
3. **Input: [7, 7, 7, 7] ,Expected Output: [7, 7, 7, 7]**
4. **Input: [-5, -1, -3, -2, -4] , Expected Output: [-5, -4, -3, -2, -1]**

**Code:**

def process\_list(input\_list):

return input\_list

test\_cases = [[],[1],[7, 7, 7, 7],[-5, -1, -3, -2, -4]]

for i, test\_case in enumerate(test\_cases, 1):

result = process\_list(test\_case)

print(result)

1. **Describe the Selection Sort algorithm's process of sorting an array. Selection Sort works by dividing the array into a sorted and an unsorted region. Initially, the sorted region is empty, and the unsorted region contains all elements. The algorithm repeatedly selects the smallest element from the unsorted region and swaps it with the leftmost unsorted element, then moves the boundary of the sorted region one element to the right. Explain why Selection Sort is simple to understand and implement but is inefficient for large datasets. Provide examples to illustrate step-by-step how Selection Sort rearranges the elements into ascending order, ensuring clarity in your explanation of the algorithm's mechanics and effectiveness.**

**Sorting a Random Array:**

**Input: [5, 2, 9, 1, 5, 6] Output: [1, 2, 5, 5, 6, 9]**

**Sorting a Reverse Sorted Array: Input: [10, 8, 6, 4, 2] Output: [2, 4, 6, 8, 10]**

**Sorting an Already Sorted Array: Input: [1, 2, 3, 4, 5] Output: [1, 2, 3, 4, 5]**

**Code:**

def selection\_sort(arr):

n = len(arr)

for i in range(n):

min\_idx = i

for j in range(i+1, n):

if arr[j] < arr[min\_idx]:

min\_idx = j

arr[i], arr[min\_idx] = arr[min\_idx], arr[i]

return arr

random\_array = [5, 2, 9, 1, 5, 6]

sorted\_random = selection\_sort(random\_array)

print(sorted\_random)

reverse\_sorted\_array = [10, 8, 6, 4, 2]

sorted\_reverse\_sorted = selection\_sort(reverse\_sorted\_array)

print(sorted\_reverse\_sorted)

already\_sorted\_array = [1, 2, 3, 4, 5]

sorted\_already\_sorted = selection\_sort(already\_sorted\_array)

print(sorted\_already\_sorted)

**3.Write code to modify bubble sort function to stop early if the list becomes sorted before all passes are completed.**

**Code:**

def bubble\_sort(arr):

n = len(arr)

for i in range(n):

swapped = False

for j in range(0, n-i-1):

if arr[j] > arr[j+1]:

arr[j], arr[j+1] = arr[j+1], arr[j]

swapped = True

if not swapped:

break

return arr

random\_array = [5, 2, 9, 1, 5, 6]

sorted\_random = bubble\_sort(random\_array)

print(sorted\_random)

reverse\_sorted\_array = [10, 8, 6, 4, 2]

sorted\_reverse\_sorted = bubble\_sort(reverse\_sorted\_array)

print(sorted\_reverse\_sorted)

already\_sorted\_array = [1, 2, 3, 4, 5]

sorted\_already\_sorted = bubble\_sort(already\_sorted\_array)

print(sorted\_already\_sorted)

**4.Write code for Insertion Sort that manages arrays with duplicate elements during the sorting process. Ensure the algorithm's behaviour when encountering duplicate values, including whether it preserves the relative order of duplicates and how it affects the overall sorting outcome. Examples: 1. Array with Duplicates: o Input: [3, 1, 4, 1, 5, 9, 2, 6, 5, 3] o Output: [1, 1, 2, 3, 3, 4, 5, 5, 6, 9] 2. All Identical Elements: o Input: [5, 5, 5, 5, 5] o Output: [5, 5, 5, 5, 5] 3. Mixed Duplicates: o Input: [2, 3, 1, 3, 2, 1, 1, 3] o Output: [1, 1, 1, 2, 2, 3, 3, 3]**

**Code:**

def insertion\_sort(arr):

n = len(arr)

for i in range(1, n):

key = arr[i]

j = i - 1

while j >= 0 and arr[j] > key:

arr[j + 1] = arr[j]

j -= 1

arr[j + 1] = key

return arr

array\_with\_duplicates = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]

sorted\_with\_duplicates = insertion\_sort(array\_with\_duplicates)

print(sorted\_with\_duplicates)

all\_identical\_elements = [5, 5, 5, 5, 5]

sorted\_identical = insertion\_sort(all\_identical\_elements)

print(sorted\_identical)

mixed\_duplicates = [2, 3, 1, 3, 2, 1, 1, 3]

sorted\_mixed\_duplicates = insertion\_sort(mixed\_duplicates)

print(sorted\_mixed\_duplicates)

**5.Given an array arr of positive integers sorted in a strictly increasing order, and an integer k. return the kth positive integer that is missing from this array. Example 1: Input: arr = [2,3,4,7,11], k = 5 Output: 9 Explanation: The missing positive integers are [1,5,6,8,9,10,12,13,...]. The 5th missing positive integer is 9. Example 2: Input: arr = [1,2,3,4], k = 2 Output: 6 Explanation: The missing positive integers are [5,6,7,...]. The 2nd missing positive integer is 6.**

**Code :**

def findKthPositive(arr, k):

missing\_count = 0

current = 1

index = 0

while True:

if index >= len(arr) or arr[index] != current:

missing\_count += 1

if missing\_count == k:

return current

else:

index += 1

current += 1

arr1 = [2, 3, 4, 7, 11]

k1 = 5

print(findKthPositive(arr1, k1))

**6.A peak element is an element that is strictly greater than its neighbors. Given a 0-indexed integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to any of the peaks. You may imagine that nums[-1] = nums[n] = -∞. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array. You must write an algorithm that runs in O(log n) time. Example 1: Input: nums = [1,2,3,1] Output: 2 Explanation: 3 is a peak element and your function should return the index number 2. Example 2: Input: nums = [1,2,1,3,5,6,4] Output: 5 Explanation: Your function can return either index number 1 where the peak element is 2, or index number 5 where the peak element is 6.**

**Code :**

def findPeakElement(nums):

left, right = 0, len(nums) - 1

while left < right:

mid = (left + right) // 2

if nums[mid] > nums[mid + 1]:

right = mid

else:

left = mid +1

return left

nums1 = [1, 2, 3, 1]

print(findPeakElement(nums1))

**7. Given two strings needle and haystack, return the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack. Example 1: Input: haystack = "sadbutsad", needle = "sad" Output: 0 Explanation: "sad" occurs at index 0 and 6. The first occurrence is at index 0, so we return 0. Example 2: Input: haystack = "leetcode", needle = "leeto" Output: -1 Explanation: "leeto" did not occur in "leetcode", so we return -1.**

**Code :**

def find\_needle(haystack, needle):

return haystack.find(needle)

haystack1 = "sadbutsad"

needle1 = "sad"

result1 = find\_needle(haystack1, needle1)

print(result1)

**8. Given an array of string words, return all strings in words that is a substring of another word. You can return the answer in any order. A substring is a contiguous sequence of characters within a string Example 1: Input: words = ["mass","as","hero","superhero"] Output: ["as","hero"] Explanation: "as" is substring of "mass" and "hero" is substring of "superhero". ["hero","as"] is also a valid answer. Example 2: Input: words = ["leetcode","et","code"] Output: ["et","code"] Explanation: "et", "code" are substring of "leetcode". Example 3: Input: words = ["blue","green","bu"] Output: [] Explanation: No string of words is substring of another string.**

**Code :**

def find\_substrings(words):

result = []

for i in range(len(words)):

for j in range(len(words)):

if i != j and words[i] in words[j]:

result.append(words[i])

break

return result

words1 = ["mass", "as", "hero", "superhero"]

output1 = find\_substrings(words1)

print(output1)